浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：范源颢

学　　院：计算机科学与技术学院

　　系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3180103574

指导教师：陈文智

2020 年　1 月 9　日

浙江大学实验报告

课程名称：计算机体系结构 实验类型：综合

实验项目名称：多周期访存的流水线 CPU

学生姓名：范源颢 专业：计算机科学技术 学号：3180103574

同组学生姓名：周寒靖 指导老师：陈文智

实验地点：曹光彪西楼-301 实验日期：2020 年 1 月 9 日

一、 实验目的和要求

目的：

1. 理解缓冲存储行的概念（Cache Line）
2. 理解缓存管理单元（Cache Management Unit, CMU）的原理和 CMU 的状态机
3. 掌握 CMU 的设计方法
4. 掌握设计缓存行的方法
5. 掌握验证缓存行的方法。

要求：

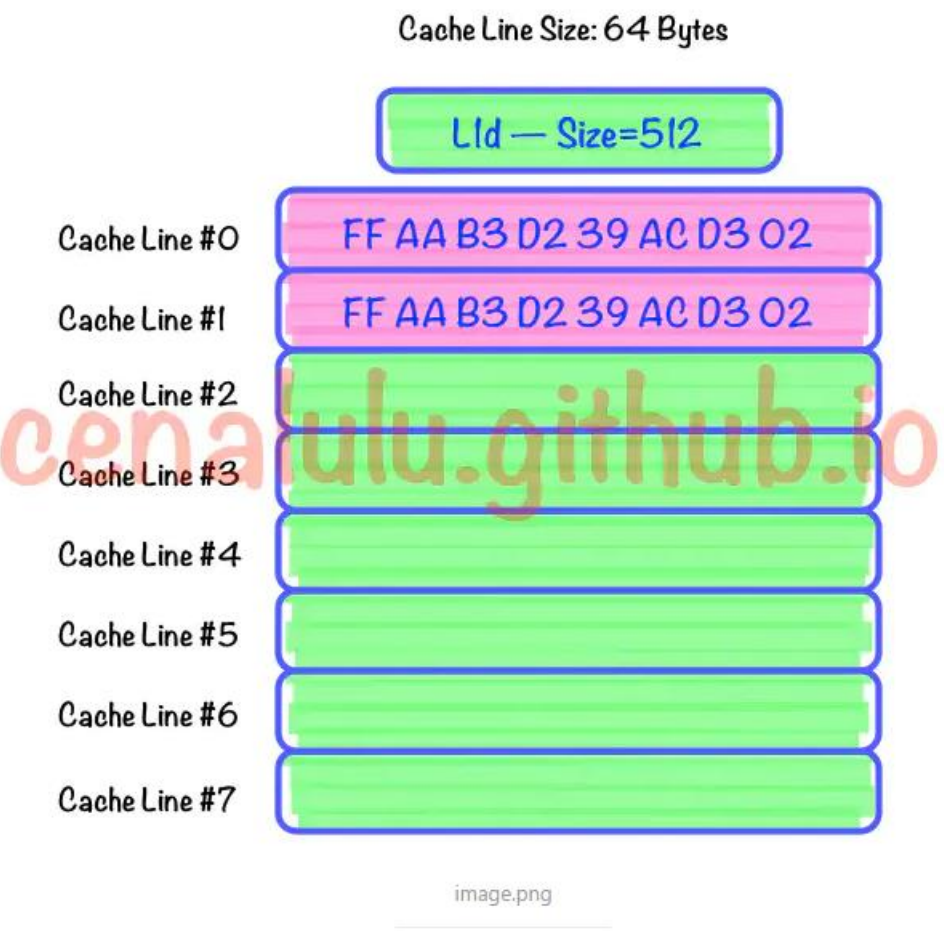
1. 设计缓存行和 CMU；
2. 验证缓存行和 CMU
3. 观察仿真波形
4. （注意，我们这次试验是不需要上板实现的，只要波形对就可以通过验收了，【这也是这个实验被作为 Bonus 的特性】）

二、 实验内容和原理

2.1 缓存行

缓存行可以理解为 CPU 缓存中最小的缓存单元，目前主流的 CPU 缓存的缓存行都是 64B 的（现在似乎已经是 64 位的天下了），假如我们有一个 512B 的一级缓存，那

么我们按照 64B 为一个缓存单位的大小来算，我们知道一个缓存可以存放 512/64 = 8 个缓存行，如下所述：



我们这里的缓存行结构将如下所示：

V	D	tag	Data

我们需要一个 V 字节（Validity）来表示此行是否有效，一个 D 字节（Dirty）来表示此行是否是脏的，以及 Tag 和 Data 来分别表示这一行的标签和数据。

2.2 缓存模式和寻址

本次实验我们实现的缓存有如下性质：

- 直接映射（不使用任何组相连【No Associativity】）
- 回写（write-back，写命中【write hit】的策略，区别于透写【write-through】，CPU 更新缓存时，仅仅更新缓存行的数据和缓存行标记，并不同步更新内存【或者更一般的，我们说这个缓存的下级存储器】，只有在这个缓存行被换出时，我们再考虑他和下级存储器的同步，这时，缓存更新了而没有下级存储器没有更新的缓存行，我们需要将“脏位”置一）
- 按写分配（write allocate，写失配【write Miss】的策略，区别于绕写【write-around】，我们在发现 CPU 写缓存失配时，即要写入的缓存行已经不再在缓存中时，我们需要直接从内存中读出需要的缓存行，然后写入，这样，写失配将和读失配一样引起缓存行的换出，这不像绕写，绕写时我们会直接写下级存储而绕过缓存，只有读失配才会引起缓存行的换出）

本次实验，我们使用的寻址方法如下：

我们在实验中，约定：一个字（word）是 4 个字节（Byte）（即是 4 * 8 = 32 bit），一个缓存行有 4 个字，而一个地址是 32 位（bit = 1/8 Byte）的，同时约定标签是 22 位的。如此一来，我们的块内偏移量（block offset）应该是 2 + 2 = 4，因为我们在用 Index 和 Tag 锁定一个缓存行的时候，我们需要 2 位行-字宽（LINE_WORDS_WIDTH）来确定这是哪个字，2 位字-字节宽（WORD_BYTE_WIDTH）来确定是哪个字节，如此一来，我们余下的 Index 是 32 – 22 – 4 = 6，那么我们可以索引的缓存大小是 2^6 = 64，也就是有 64 个缓存行的缓存。

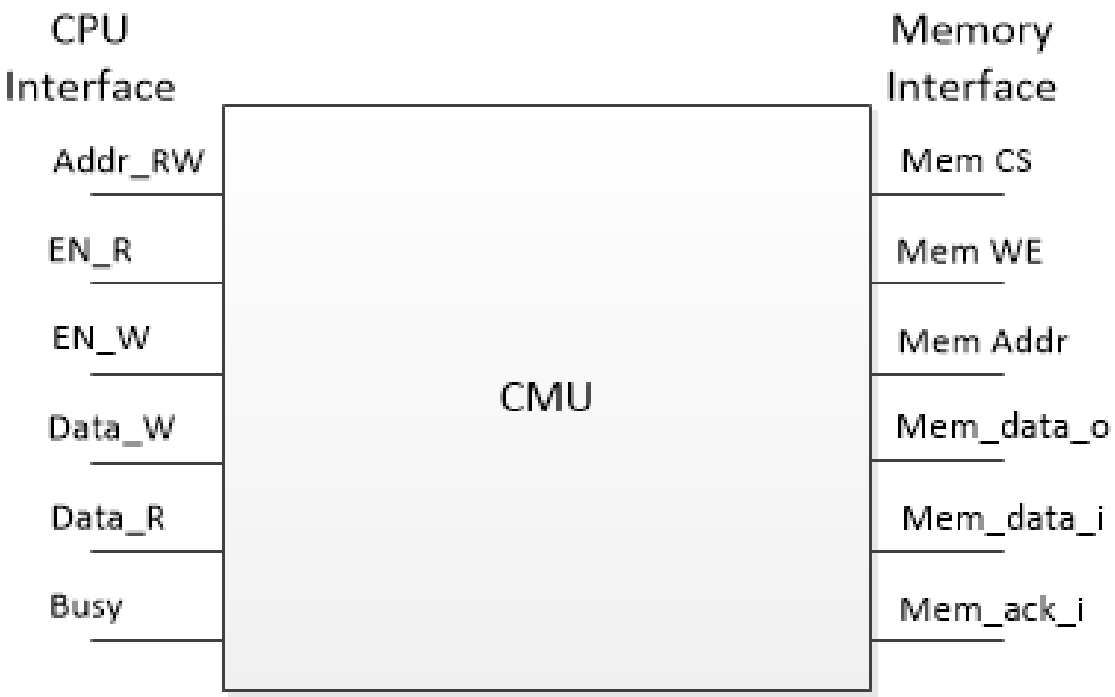
2.3 缓存行和缓存的设计

我们使用参数（parameter）来表示上文所述的变量

```
parameter
        WORD_BITS = 32,
        ADDR_BITS = 32,
        TAG_BITS = 22,
        WORD_BYTES_WIDTH = 2,
        LINE_WORDS_WIDTH = 2,
        LINE_WORDS = 4
        LINE_NUM = 64,
        LINE_INDEX_WIDTH = 6;
```

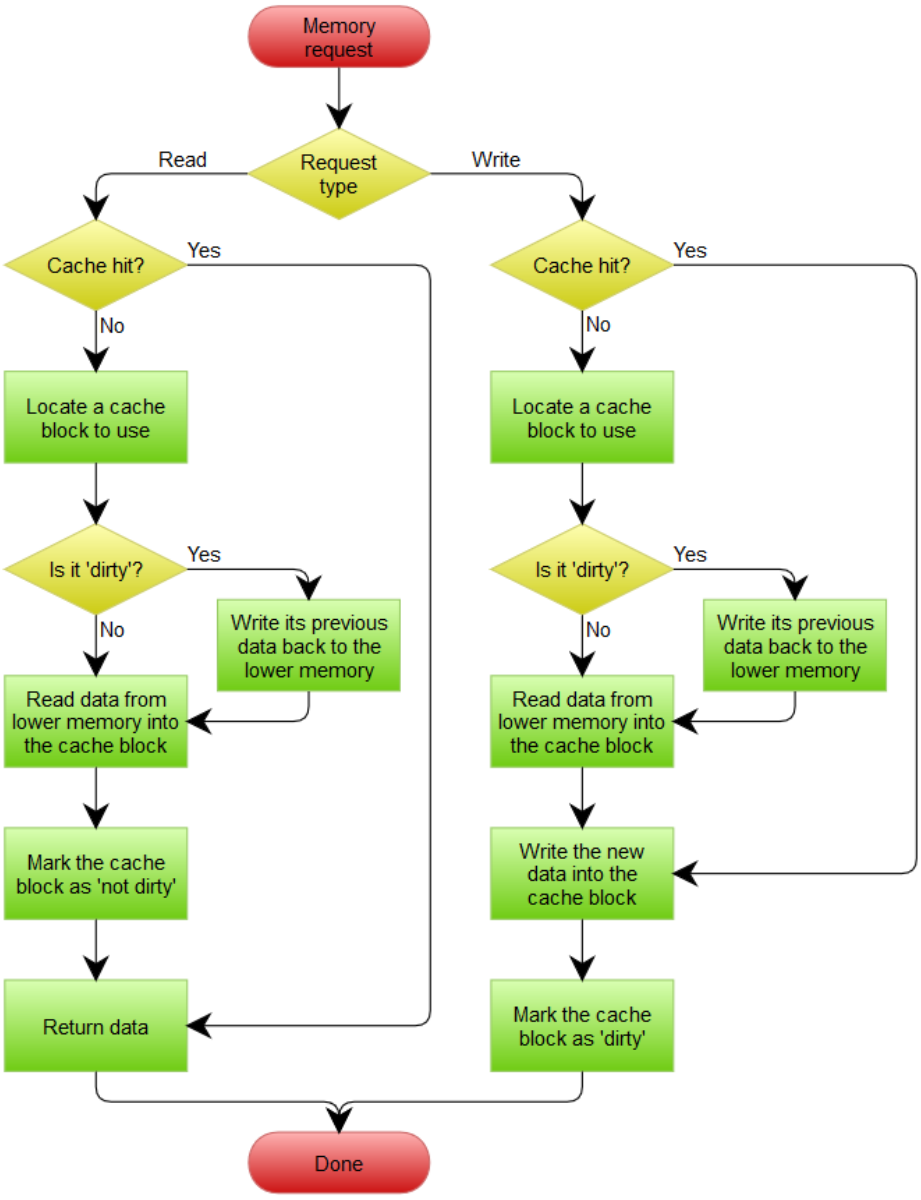
之后，我们设计两个 64（ = LINE_NUM）位的元素 inner_valid，inner_dirty，用于实现有效位和脏位的表示。然后我们需要 64（= LINE_NUM）个，22（= TAG_BITS）位的标签数组，inner_tag； 最后我们需要 32（= WORD_BITS）位的，128（= LINE_NUM * LINE_WORDS）个的数据数组，inner_data，之所以要乘上一个 LINE_WORDS ,是因为按照之前约定，一个缓存行中应当有 4 个字，那么我们在使用 tag、index 来标记位置之后，还需要一个 word_offset 来指明是哪个字。

2.4 CMU 的设计：



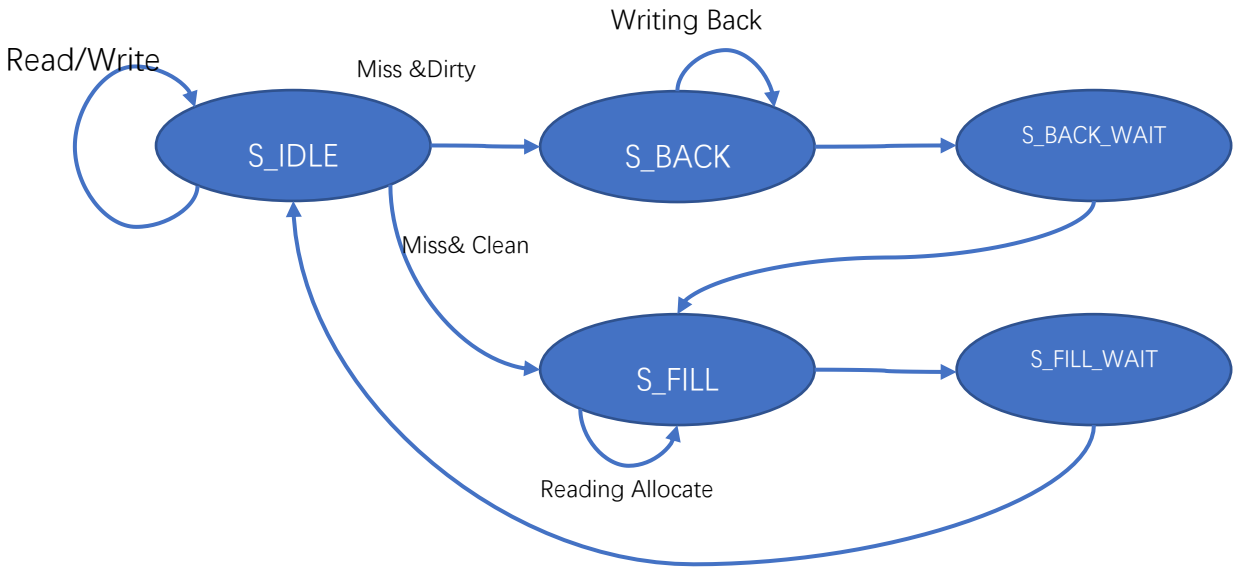
CMU 是 CPU 和存储单元的接口，我们需要通过 CMU 进行访存使得访存的速度提高，而 CMU 中自然包含 cache，因为 cache 是提高访存速度的关键。

CMU 的工作流程可以用如下的流程图来表示，我们关心的，无非是六种情况，读出时，我们的命中和失配是两种，写入时，我们的命中和失配也是两种，我们还有换出的策略，换出时遇到脏行和不是脏行又是两种，注意我们的按写分配的策略，在读出和写入的过程中都有可能引发换出，因此，我们会有如下的流程图作为总结：



2.5 CMU 的有限状态机表示：

我们可以用我们的有限状态机图来表示我们的 CMU；



S_IDLE：空闲状态，不进行 Memory 操作。

S_BACK：将数据写回到 Memory，由计数器控制直到整个 Cache Line 全部写回。

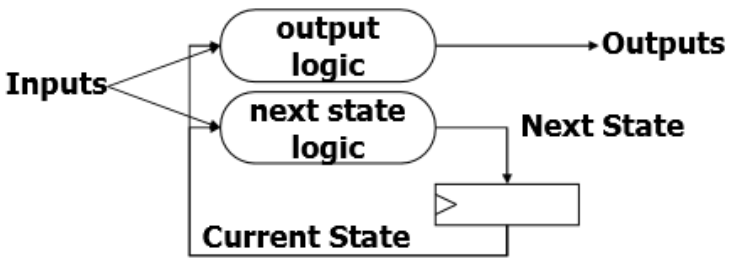
S_BACK_WAIT：等待一个时钟，需要释放对 Memory 的片选以准备下次操作。

S_FILL：从 Memory 读取数据，由计数器控制直到整个 Cache Line 全部读取完毕。

S_FILL_WAIT：等待一个时钟，保证最后一个数据成功写入 Cache。

Miss & Dirty 表示此时一个已经在缓存中被更新的脏行需要被换出，于是我们需要先写回（进入 S_BACK 状态），然后在读入新的行（S_BACK_WAIT 后进入 S_FILL）。假如情况是 Miss & Clean，那么我们需要换出的行并没有在缓存中被更新，于是我们仅仅从内存中读取新的行（进入 S_FILL）即可。

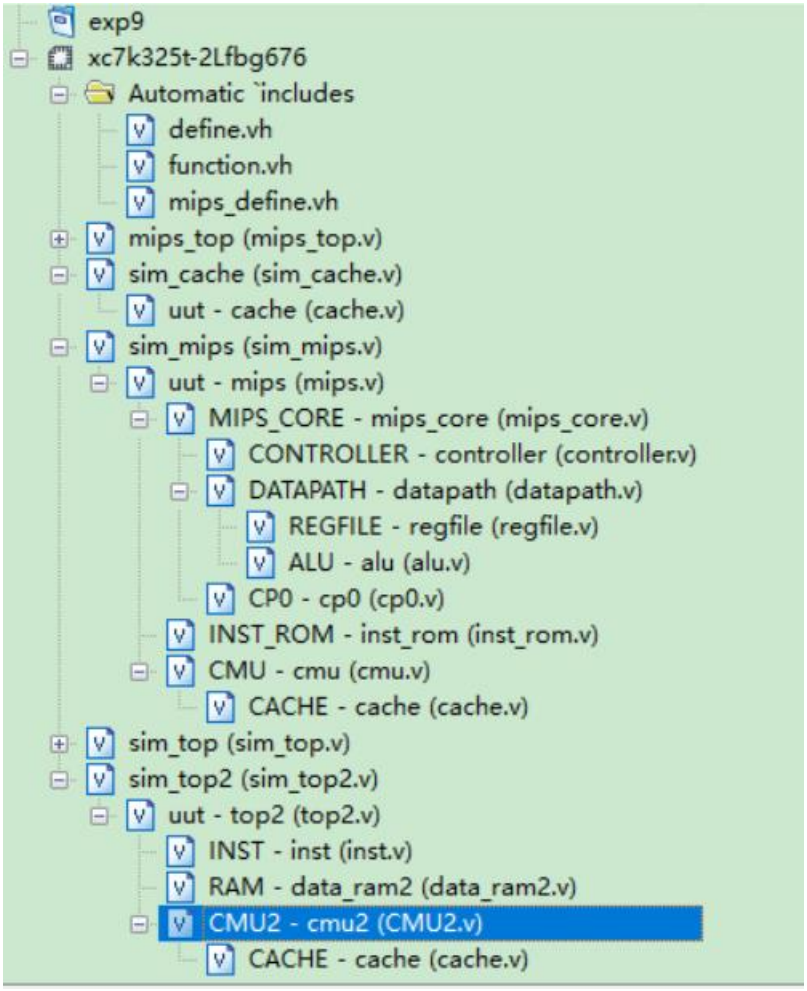
我们在数字逻辑设计中，已经对有限状态机的概念有所了解了，我们的有限状态机可以使用下图所示的模型表示：



关键是状态转移的逻辑，亦即 next state logic，还有输出的部分（output）

三、 实验过程和数据记录

实验的文件结构如下：



Automatic includes 文件夹下定义了我们可以在代码中使用的头文件；

我们书写了一个单一的文件 sim_cache 用于仿真 cache 的效果，置于 CMU 则比较复杂，我们首先实现（从 PPT 中参考）了另外两个模块 INST 和 RAM，分别用作仿真的 CPU 指令和数据集，之后将这些模块组装为 top，再使用 sim_top 对 top 模块仿真。

3.1 缓存：

缓存的程序接口：

```
module cache (
 input wire clk, // clock
 input wire rst, // reset
 input wire [ADDR_BITS-1:0] addr, // address
 input wire store, // set valid to 1 and reset dirty to 0
 input wire edit, // set dirty to 1
 input wire invalid, // reset valid to 0
 input wire [WORD_BITS-1:0] din, // data write in
```

```
    output wire hit, // hit or not
    output wire [WORD_BITS-1:0] dout, // data read out
    output wire valid, // valid bit
    output wire dirty, // dirty bit
    output wire [TAG_BITS-1:0] tag // tag bits
);
```

其中关键的输入是 addr，也就是执行单元 CPU 输入的寻址信号,在写时，我们还需要给出 din 信号表示写的数据，剩余的 store，edit，invalid 是输入的控制信号，我们应该给出的数据信号时 dout，表示我们读出的数据为何，之后反馈的 valid、dirty、tag 信号用于判断我们的读写是否正确。

我们读出的数据只需要在按照我们之前的约定，取 Index 和 LINE_WORDS_WIDTH 之间的值作为索引即可，我们把他写在下文中的时钟周期里面

而如果要写一个缓存行，那么我们需要的条件是：或者 store 信号置位，表示写入一项新的缓存行（同时 store 也在指示有效位变为 1，脏位变为 0，如接口处的注释所言），或者 edit 和 hit 同时置位，表示我们在回写，写入了一个脏的内存页。

之后我们把 invalid 和 store、edit 的逻辑全部整合起来，就得到了如下：

```
always @(posedge clk) begin
        dout <= inner_data[addr[ADDR_BITS-TAG_BITS-1:WORD_BYTES_WIDTH]]
;
        if (rst) begin
            inner_valid <= 0;
                inner_dirty <= 0;
        end else begin
            if (store || (edit && hit)) begin
                inner_data[addr[ADDR_BITS-TAG_BITS-1:WORD_BYTES_WIDTH]]
 <= din;
            end
            if (invalid) begin
                inner_valid[addr[ADDR_BITS-TAG_BITS-1:LINE_WORDS_WIDTH+
WORD_BYTES_WIDTH]] <= 0;
                inner_dirty[addr[ADDR_BITS-TAG_BITS-1:LINE_WORDS_WIDTH+
WORD_BYTES_WIDTH]] <= 0; //TODO not sure,should be 0,dirty bit is at le
ast modified once
            end else if(store) begin
                inner_valid[addr[ADDR_BITS-TAG_BITS-1:LINE_WORDS_WIDTH+
WORD_BYTES_WIDTH]] <= 1;
                inner_dirty[addr[ADDR_BITS-TAG_BITS-1:LINE_WORDS_WIDTH+
WORD_BYTES_WIDTH]] <= 0;
```

```
                inner_tag[addr[ADDR_BITS-TAG_BITS-1:LINE_WORDS_WIDTH+WO
RD_BYTES_WIDTH]] <= addr[ADDR_BITS-1:ADDR_BITS-TAG_BITS];
            end else if (edit) begin
                inner_dirty[addr[ADDR_BITS-TAG_BITS-1:LINE_WORDS_WIDTH+
WORD_BYTES_WIDTH]] <= 1;
                inner_tag[addr[ADDR_BITS-TAG_BITS-1:LINE_WORDS_WIDTH+WO
RD_BYTES_WIDTH]] <= addr[ADDR_BITS-1:ADDR_BITS-TAG_BITS];//TODO seems n
o need to set tag?
            end
        end
    end
```

最后我们在处理需要输出的 valid、 dirty、tag，另外，我们还需要处理 hit 的意义：

```
assign hit = (addr[ADDR_BITS-1:ADDR_BITS-TAG_BITS]==inner_tag[addr[ADDR
_BITS-TAG_BITS-1:LINE_WORDS_WIDTH+WORD_BYTES_WIDTH]]) && valid;
assign valid = inner_valid[addr[ADDR_BITS-TAG_BITS-1:LINE_WORDS_WIDTH+W
ORD_BYTES_WIDTH]];
assign dirty = inner_dirty[addr[ADDR_BITS-TAG_BITS-1:LINE_WORDS_WIDTH+W
ORD_BYTES_WIDTH]];
assign tag = inner_tag[addr[ADDR_BITS-TAG_BITS-1:LINE_WORDS_WIDTH+WORD_
BYTES_WIDTH]];
```

至此我们完成了 cache 的实现，他的管理依赖于 cmu(cache manage unit)

3.2 CMU

3.2.1 CMU 的实现-接口和状态转移逻辑：

CMU 的接口如下：

我们主要根据 2.5 的状态图设计我们的状态转移逻辑，按照 PPT，我们书写如下：

```
            S_IDLE: begin
            if (en_r || en_w) begin
                if (cache_hit)
                    next_state = S_IDLE;
                else if (cache_valid && cache_dirty)
                    next_state = S_BACK;
                else
                    next_state = S_FILL;
            end
        end
        S_BACK: begin
            if (mem_ack_i)
                next_word_count = word_count + 1'h1;
```

```
                else
                    next_word_count = word_count;
                if (mem_ack_i && word_count == {LINE_WORDS_WIDTH{1'b1}}
)
                    next_state = S_BACK_WAIT;
                else
                    next_state = S_BACK;
            end
            S_BACK_WAIT: begin
                next_word_count = 0;
                next_state = S_FILL;
            end
            S_FILL: begin
                if (mem_ack_i)
                    next_word_count = word_count + 1'h1;
                else
                    next_word_count = word_count;
                if (mem_ack_i && word_count == {LINE_WORDS_WIDTH{1'b1}}
)
                    next_state = S_FILL_WAIT;
                else
                    next_state = S_FILL;
            end
            S_FILL_WAIT: begin
                next_word_count = 0;
                next_state = S_IDLE;
            end
```

S_IDLE 状态下，我们等待 en_r 和 en_w 信号，也就是 CPU 的读写，此时假若有缓存命中信号（cache_hit），我们保持在这一状态（S_IDLE），假若读到或写到脏行，那么跳转到 S_BACK 状态，假如不是脏行，或者此行无效，我们需要跳转到 S_FILL。

我们之后考虑 S_BACK 状态（S_FILL 同理），这个状态下，我们首先判断 mem_ack_i 信号，也就是判断是否有一个字被成功从内存读出，若有，我们下一个周期需要读出的字就是当下读出的字加一（next_word_count = word_count + 1’h1），否则，我们保持需要读出的字为上个周期的字，假如我们已经读完四个字（mem_ack_i && word_count == {LINE_WORDS_WIDTH{1'b1}}），那么我们需要跳转到 S_BACK_WAIT 状态，假如没有到达四个字，那么保持 S_BACK 状态。

之后我们考虑 S_BACK_WAIT 状态（S_FILL_WAIT 同理），这个状态下，我们仅仅将 next_word_count 置位为 0，因为此时一个向下一级操作内存行的逻辑已经结束，下一次我们

取内存行一定是从第 0 个开始取（取到第 3 个，总共 4 个），所以我们需要对 next_word_count 信号进行处理，最后当然还要将 next_state 跳转为 S_IDLE。

当然，不要忘了一个控制跳转的逻辑，处理下一个时钟周期的状态（state）和应操作字（word_count），我们使用时钟上沿的信号进行同步：

```
always @(posedge clk) begin
    if (rst) begin
        state <= 0;
        word_count <= 0;
    end
    else begin
        state <= next_state;
        word_count <= next_word_count;
    end
end
```

3.2.2 CMU 的实现-输出信号控制：

我们按照 PPT 的提示，首先，不再按照逐一列举的方法处理各个状态的输出信号，而是尽可能将输出信号相同的状态合并，其次，我们将输出信号划分为两组，对 cache 的操作，对 memory 的操作，在两个 always @(posedge clk)begin end 语句块中处理。

对于缓存的控制逻辑如下：

```
// cache control
    reg [LINE_WORDS_WIDTH-1:0] word_count_buf;
    always @(posedge clk) begin
        if (rst)
            word_count_buf <= 0;
        else
            word_count_buf <= word_count;
    end

    always @(*) begin
        cache_store = 0;
        cache_edit = 0;
        cache_addr = 0;
        cache_din = 0;
        case (next_state)
            S_IDLE: begin
                cache_addr = addr_rw;
                cache_edit = en_w;
                cache_din = data_w;
            end
            S_BACK, S_BACK_WAIT: begin
```

```
                cache_addr = {addr_rw[31:LINE_WORDS_WIDTH+2], next_word
_count, 2'b00};
            end
            S_FILL, S_FILL_WAIT: begin
                cache_addr = {addr_rw[31:LINE_WORDS_WIDTH+2], word_coun
t_buf, 2'b00};
                cache_din = mem_data_i;
                cache_store = mem_ack_i;
            end
        endcase
    end
```

这里注意，我们在 S_IDLE 状态下，可以单纯的使用 CPU 给出的地址（input 信号 addr_rw）进行寻址，因为 cache 会负责处理 addr_rw 信号的解码，但是我们不应该在 S_BACK(_WAIT)和 S_FILL(_WAIT)的状态下使用 addr_rw 信号，我们要读取的是之前提到的 next_word_count 的信号，所以我们的信号将如上所示。另外，注意 BACK 状态是写内存读缓存，所以我们只要单纯的给出地址信号即可，而 FILL 状态是读内存写缓存，于是我们需要再给出 cache_din 信号，赋值为 mem_data_i 信号，作为内存的给缓存的输入，而 cache_store 信号则用于指示写使能，之所以用 mem_ack_i 信号赋值，是因为在 mem_ack_i 置位时，内存读出的信号才是有效的。

另一方面，对内存的控制逻辑如下：

```
always @(posedge clk) begin
        mem_cs_o <= 0;
        if (rst) begin
            mem_we_o <= 0;
            mem_addr_o <= 0;
        end
        else case (next_state)
            S_IDLE, S_BACK_WAIT, S_FILL_WAIT: begin
                mem_cs_o <= 0;
                mem_we_o <= 0;
                mem_addr_o <= 0;
            end
            S_BACK: begin
                mem_cs_o <= 1;
                mem_we_o <= 1;
                mem_addr_o <= {cache_tag, addr_rw[31-TAG_BITS:LINE_WORD
S_WIDTH+2], next_word_count, 2'b00};
            end
            S_FILL: begin
                mem_cs_o <= 1;
```

```
                mem_we_o <= 0;
                mem_addr_o <= {addr_rw[31:LINE_WORDS_WIDTH+2], next_wor
d_count, 2'b00};
            end
        endcase
    end
```

我们在 S_BACK 和 S_FILL 的状态下给出了片选信号，按照我们上次处理逻辑，只有片选开始，内存才会开始读取，而过了大约 7 个时钟周期，我们才会得到需要的值，这时 mem_ack_i 会置位，我们根据上文中缓存的控制逻辑可以直接让缓存读取，而不需要再在内存控制逻辑上做些什么了，内存的逻辑只在 S_BACK 和 S_FILL 给出片选信号，并且指定我们需要读/写的地址。S_BACK 是读缓存写内存，于是我们还需要给出写使能信号（mem_we_o，for memory write enable output）。

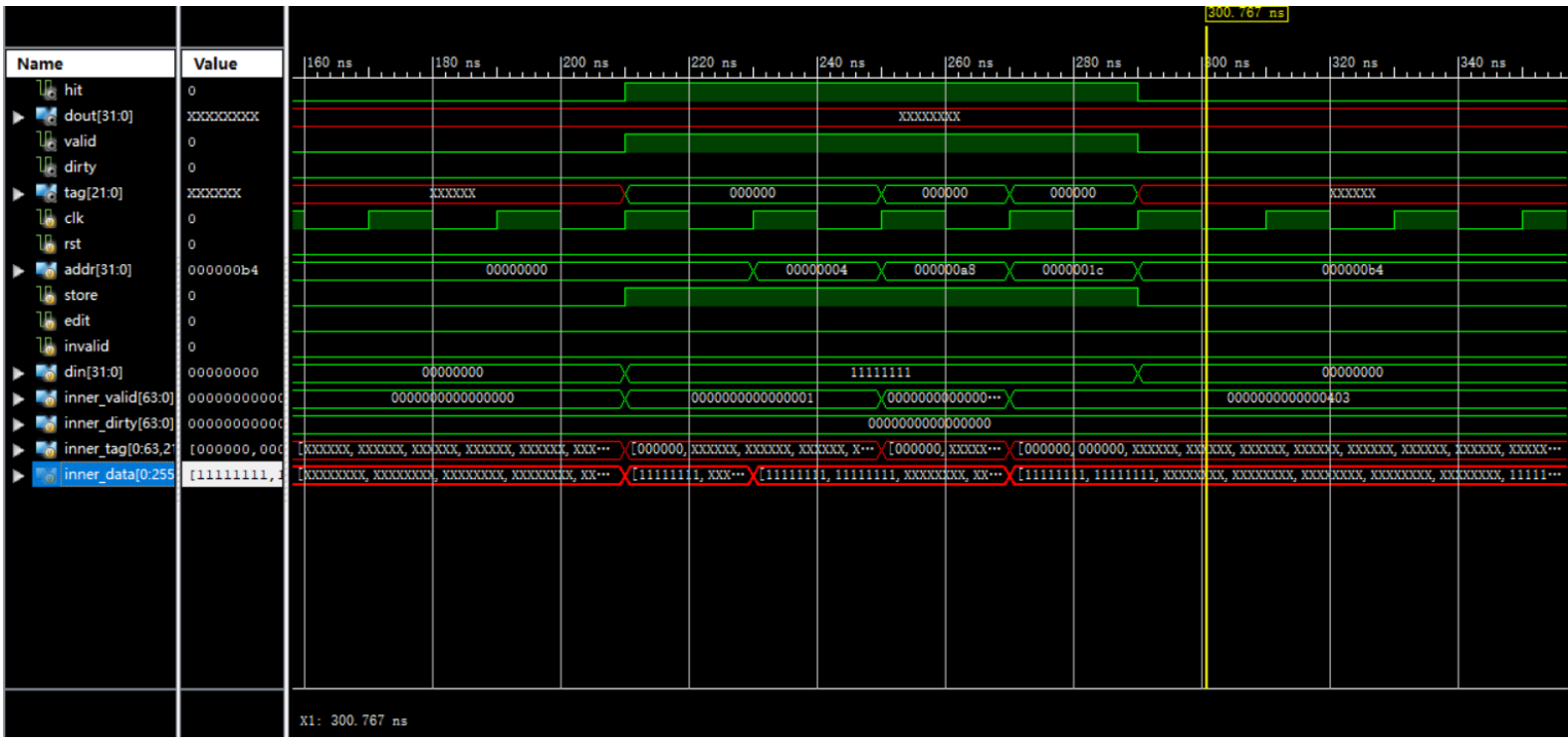
最后不要忘记给变量定义和声明，以及连接 cache，mem_data_o、data_r 和 cache_dout 保持一致即可，而 stall 信号则在下一个状态并非 S_IDLE 时置位，使得 CMU 运作时暂停其他模块的工作。

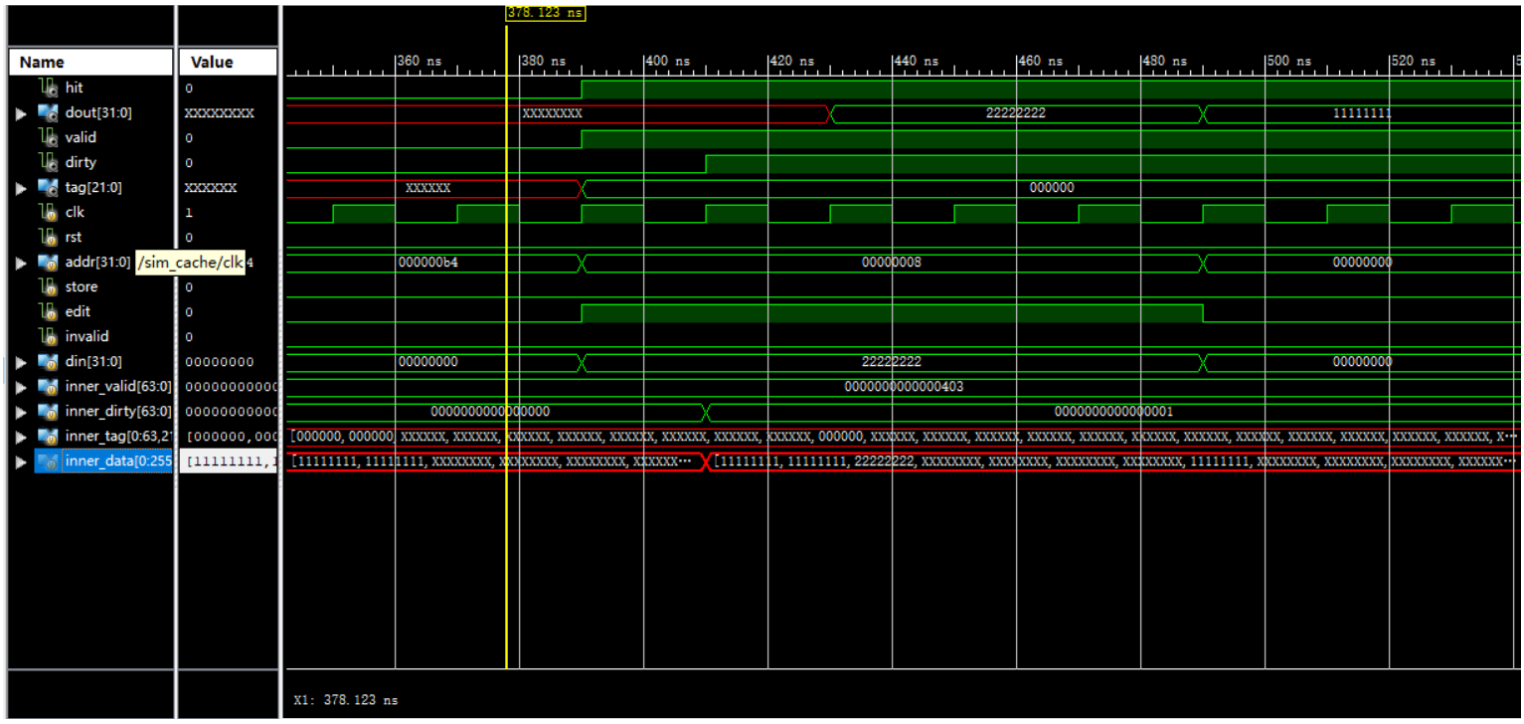
```
    // cache core
    reg cache_store;
    reg cache_edit;
    reg [31:0] cache_addr;
    reg [31:0] cache_din;
    wire [31:0] cache_dout;
    wire [TAG_BITS-1:0] cache_tag;
    wire cache_hit, cache_valid, cache_dirty;

    cache #(
        .TAG_BITS(TAG_BITS),
        .LINE_WORDS(LINE_WORDS),
        .LINE_WORDS_WIDTH(LINE_WORDS_WIDTH)
        ) CACHE (
        .clk(clk),
        .rst(rst),
        .addr(cache_addr),
        .store(cache_store),
        .edit(cache_edit),
        .invalid(1'b0),
        .din(cache_din),
        .hit(cache_hit),
        .dout(cache_dout),
        .valid(cache_valid),
        .dirty(cache_dirty),
```

```
        .tag(cache_tag)
    );

    localparam
        S_IDLE = 0,
        S_BACK = 1,
        S_BACK_WAIT = 2,
        S_FILL = 3,
        S_FILL_WAIT = 4;

    reg [2:0] state = 0;
    reg [2:0] next_state;
    reg [LINE_WORDS_WIDTH-1:0] word_count = 0;
    reg [LINE_WORDS_WIDTH-1:0] next_word_count;

    assign data_r = cache_dout;
    always @(*) begin
        mem_data_o = cache_dout;
    end

    assign stall = (next_state != S_IDLE);
```

四、 实验结果分析

4.1 cache 仿真结果：





我们在图片右侧的紫框中找到相应的芯片元件，拖入左侧的图中就可以得到输入输出的信号波形图，注意设定仿真时长，并且显示方式为 16 进制(便于阅读)。

主要观察的波形是 dout 信号的输出。

4.2 Cache 测试用程序介绍

我们测试用的程序如下（注意和之前的程序不同）：

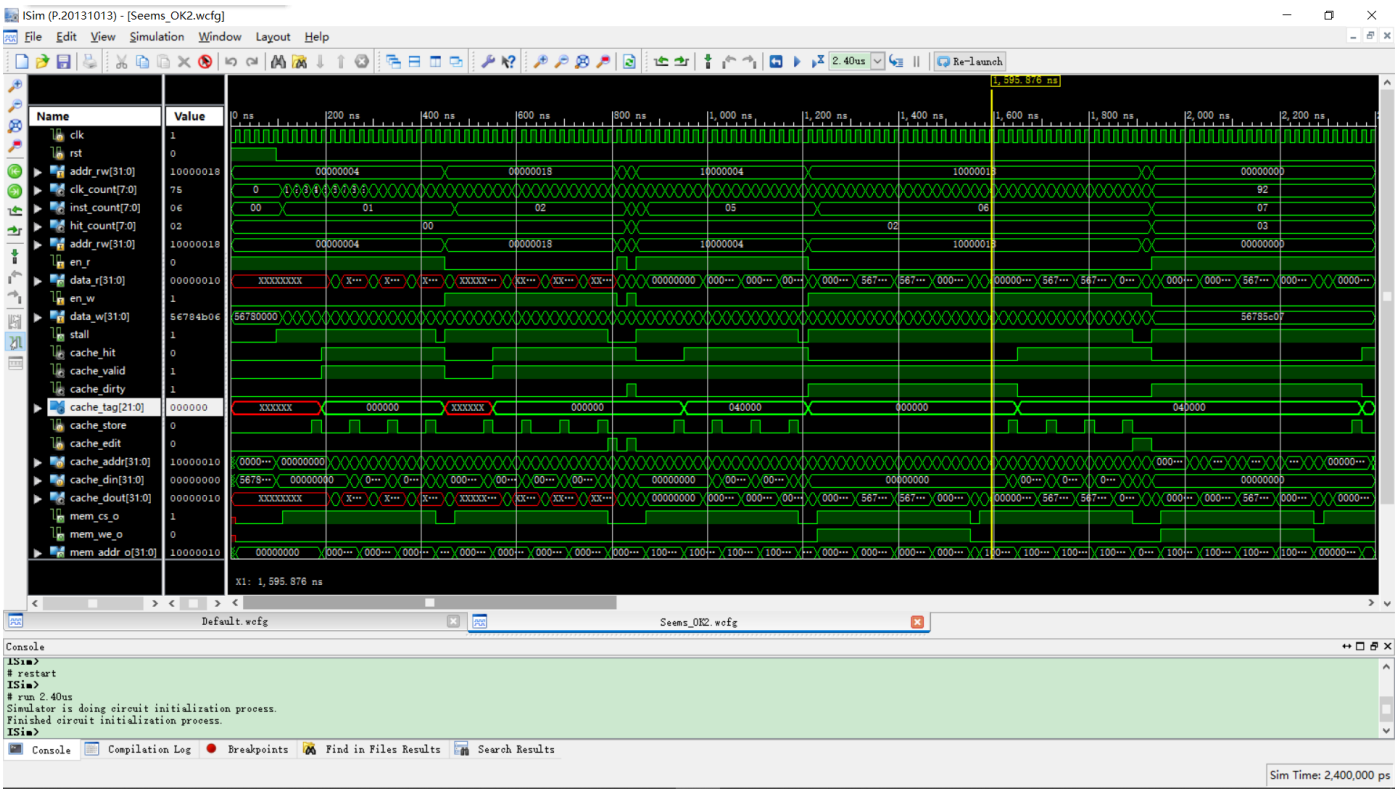
```
`timescale 1ns / 1ps

module sim_cache;

    // Inputs
    reg clk;
    reg rst;
    reg [31:0] addr;
    reg store;
    reg edit;
    reg invalid;
    reg [31:0] din;

    // Outputs
    wire hit;
    wire [31:0] dout;
    wire valid;
```

```
    wire dirty;
    wire [21:0] tag;

    // Instantiate the Unit Under Test (UUT)
    cache uut (
        .clk(clk),
        .rst(rst),
        .addr(addr),
        .store(store),
        .edit(edit),
        .invalid(invalid),
        .din(din),
        .hit(hit),
        .dout(dout),
        .valid(valid),
        .dirty(dirty),
        .tag(tag)
    );

    initial begin
        // Initialize Inputs
        rst = 0;
        addr = 0;
        store = 0;
        edit = 0;
        invalid = 0;
        din = 0;
        clk = 0;

        // Wait 100 ns for global reset to finish
//      #100;

        // Add stimulus here
//      #100;
        #210;
        store = 1;
        din = 32'h11111111;
        addr = 32'h00000000;
        #20 addr = 32'h00000004;
        #20 addr = 32'h000000a8;
        #20 addr = 32'h0000001C;
        #20;
        store = 0;
        addr = 32'h000000b4;;
```

```
        din = 0;
        #100;
        edit = 1;
        din = 32'h22222222;
        addr = 32'h00000008;
        #100;
        edit = 0;
        din = 0;
        addr = 0;

    end

    initial forever #10 clk = ~clk;

endmodule
```

我们直接使用的仿真代码，最后一行的 clk = ~clk 表示周期变化的时钟信号不需多言，我们看第一次判定点，是 210ns 之后的 store 指令，这时我们的目标是把 h11111111 这一 32 位的数字存入到缓存中的三个位置，分别是 00000004,000000a8,0000001C，所对应分别是：0 号位的第 1 个字；a 号位的第 2 个字，1 号位的第 2 个字的第 2 个字节，之后等待一百秒之后我们对 edit 置位，表示我们要修改其中一个内存行，具体而言，我们要把 00000008，也就是 0 号位的第 2 个字进行修改，改为 22222222，这会造成一次命中，因为二者的 tag 是一样的，而且有相同的 index，为此，我们会在之后的周期读出写入的 22222222

4.3 CMU 仿真结果展示：



我们的仿真代码有如下几个部分：

1. INST 部件【指令读取】

```
module inst (
    input wire clk,
    input wire rst,
    input wire [3:0] index,  // instruction index
    output wire valid,  // stop running if valid is 0
    output wire write,  // write enable signal for cache
    output wire [31:0] addr  // address for cache
    );

    reg [33:0] data [0:7];

    initial begin  // clock cycles are only for reference
        data[0] = 34'h200000004;  // read miss             1+18
        data[1] = 34'h300000018;  // write miss            1+18
        data[2] = 34'h200000008;  // read hit              1
        data[3] = 34'h300000014;  // write hit             1
        data[4] = 34'h210000004;  // read clean replace    1+18
        data[5] = 34'h310000018;  // write dirty replace   1+18*2
        data[6] = 34'h310000008;  // write hit             1
        data[7] = 34'h0;
    end

    assign
        valid = data[index][33],
        write = data[index][32],
        addr = data[index][31:0];

endmodule
```

2. data_ram 部件：

```
`timescale 1ns / 1ps
module data_ram2(
    input wire clk,
    input wire rst,
    input wire cs,
    input wire we,
    input wire [31:0] addr,
    input wire [31:0] din,
    output reg [31:0] dout,
    output reg stall,
    output reg ack
    );
```

```
parameter
    ADDR_WIDTH = 5,
    CLK_DELAY = 8;

reg [31:0] data [0:(1<<ADDR_WIDTH)-1];
reg [31:0] addr_buf = 0;
reg [3:0] delay_count = 0;

genvar i;
generate for (i=0; i<(1<<ADDR_WIDTH); i=i+1) begin: RAM_INIT
    initial data[i] = i;
end
endgenerate

always @(posedge clk) begin
    if (rst)
        addr_buf <= 0;
    else
        addr_buf <= addr;
end

localparam
    S_IDLE = 0,
    S_READ = 1,
    S_WRITE = 2;

reg [1:0] state = 0;

always @(posedge clk) begin
    dout <= 0;
    ack <= 0;
    state <= S_IDLE;
    if (rst || ~cs) begin
        delay_count <= 0;
    end
    else case (state)
        S_IDLE: begin
            delay_count <= 0;
            if (we)
                state <= S_WRITE;
            else
                state <= S_READ;
        end
```

```
            S_READ: begin
                if (delay_count == CLK_DELAY-2) begin
                    dout <= data[addr];
                    ack <= 1;
                end
                else if (addr_buf != addr) begin
                    delay_count <= 0;
                    state <= S_READ;
                end
                else begin
                    delay_count <= delay_count + 1'h1;
                    state <= S_READ;
                end
            end
            S_WRITE: begin
                if (delay_count == CLK_DELAY-2) begin
                    data[addr] <= din;
                    ack <= 1;
                end
                else if (addr_buf != addr) begin
                    delay_count <= 0;
                    state <= S_WRITE;
                end
                else begin
                    delay_count <= delay_count + 1'h1;
                    state <= S_WRITE;
                end
            end
        endcase
    end

    always @(*) begin
        stall = cs & (~ack);
    end

endmodule
```

这份代码和上次实验的 data_ram 相比有一些变化，我们修改了接口使得他可以和这次实验的 top 兼容，另外，我们的实现方法使用了状态机的逻辑，这是我们从这一次实验中学习到的，我们实现的功能和上一次实验的要求一样，每次接收到 CPU 给出的读写信号时，我们会等待 8 个周期才会给出需要的值。

3. top 模块

```
`timescale 1ns / 1ps
```

```
module top2 (
    input wire clk,
    input wire rst,
    output reg [7:0] clk_count = 0,
    output reg [7:0] inst_count = 0,
    output reg [7:0] hit_count = 0
    );

    // instruction
    reg [3:0] index = 0;
    wire valid;
    wire write;
    wire [31:0] addr;
    wire stall;

    inst INST (
        .clk(clk),
        .rst(rst),
        .index(index),
        .valid(valid),
        .write(write),
        .addr(addr)
    );

    always @(posedge clk) begin
        if (rst)
            index <= 0;
        else if (valid && ~stall)
            index <= index + 1'h1;
    end

    // ram
    wire mem_cs;
    wire mem_we;
    wire [31:0] mem_addr;
    wire [31:0] mem_din;
    wire [31:0] mem_dout;
    wire mem_ack;

    data_ram2 #(
        .ADDR_WIDTH(5),
        .CLK_DELAY(3)
        ) RAM (
        .clk(clk),
```

```
        .rst(rst),
        .addr({26'b0, mem_addr[5:0]}),
        .cs(mem_cs),
        .we(mem_we),
        .din(mem_din),
        .dout(mem_dout),
        .stall(),
        .ack(mem_ack)
        );

    // cache
    cmu2 CMU2 (
        .clk(clk),
        .rst(rst),
        .addr_rw(addr),
        .en_r(~write),
        .data_r(),
        .en_w(write),
        .data_w({16'h5678, clk_count, inst_count}),
        .stall(stall),
        .mem_cs_o(mem_cs),
        .mem_we_o(mem_we),
        .mem_addr_o(mem_addr),
        .mem_data_i(mem_dout),
        .mem_data_o(mem_din),
        .mem_ack_i(mem_ack)
    );

    // counter
    reg stall_prev;

    always @(posedge clk) begin
        if (rst)
            stall_prev <= 0;
        else
            stall_prev <= stall;
    end

    always @(posedge clk) begin
        if (rst) begin
            clk_count <= 0;
            inst_count <= 0;
            hit_count <= 0;
        end
```

```
        else if (valid) begin
            clk_count <= clk_count + 1'h1;
            inst_count <= index + 1'h1;
            if (~stall_prev && ~stall)
                hit_count <= hit_count + 1'h1;
        end
    end

endmodule
```

我们可以看到，这个模块是仿真主要的工作模块，他需要的信号时 rst 信号和 clk 信号，而输出的信号时 clk_count、inst_count 和 hit_count，用于给时钟周期计数，给指令计数，还有给缓存的命中次数计数。同时，他的内容描述了我们会如何组织指令集、缓存、内存的工作。

开始时，假如没有停顿，并且上一条 INST 部件的指令是有效的（valid），那么我们就直接读取下一条指令（index ++），产生 index 信号，这一信号传送给 inst 模块，这个模块内部已经存储了 8 个 34 位的 data（可以把 data 理解为一个取用数据的地址数组，因为最终 data 的后 32 位会作为 addr 信号输出给 CMU，用于从 data_ram 中取用数据）

我们看 inst 最终会给出的 8 个地址行：

```
initial begin  // clock cycles are only for reference
        data[0] = 34'h200000004;  // read miss             1+18
        data[1] = 34'h300000018;  // write miss            1+18
        data[2] = 34'h200000008;  // read hit              1
        data[3] = 34'h300000014;  // write hit             1
        data[4] = 34'h210000004;  // read clean replace    1+18
        data[5] = 34'h310000018;  // write dirty replace   1+18*2
        data[6] = 34'h310000008;  // write hit             1
        data[7] = 34'h0;
    end
```

每个地址共有 9 个 16 进制字符，于是总共是 36bit，但是我们注意到限定了是 34bit 的变量，于是他的前两位被截断（但反正第一个十六进制位都小于 4，那么对应的二进制位前两位一定都是 0），于是，以 data[0]为例，他的意义是“有效操作”(data[0][33] = 1)，“读操作”（data[0][32] = 0），“读取地址为 00000004”，结合上文所述的地址的意义，他是指 0 号标签，0 号位，第 2 个字，当然，由于是冷启动，这个读取肯定是失败的，

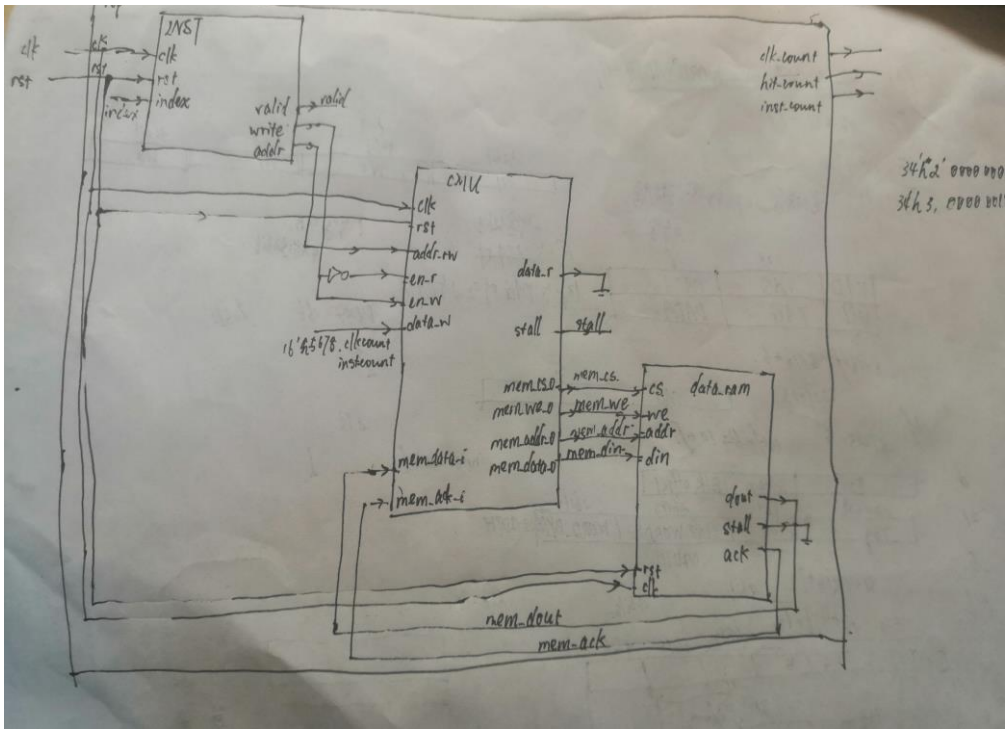
同理，我们可以有：

项目	内容	有效位	操作	地址	含义	是否命中
data[0]	34’h2_0000_0004	有效操作	读	0000_0004	0 号第 1 个字，0 标签	否
data[1]	34’h3_0000_0018	有效操作	写	0000_0018	1 号第 2 个字，	否

					0 标签	
data[2]	34’h2_0000_0008	有效操作	读	0000_0008	0 号第 2 个字，0 标签	命中 data[0]
data[3]	34’h3_0000_0014	有效操作	写	0000_0014	1 号第 1 个字，0 标签	命中 data[1],相应缓存行脏化
data[4]	34’h2_1000_0004	有效操作	读	1000_0004	0 号第 1 个字，1 标签	和 data[2]冲突,需要重新读入数据
data[5]	34’h3_1000_0018	有效操作	写	1000_0018	1 号第 2 个字，1 标签	和脏行 data[3]冲突，需要将 data[3]写回内存，同时重新读入数据
data[6]	34’h3_1000_0008	有效操作	写	1000_0008	0 号第 2 个字，1 标签	命中 data[4],不需要重新读入
data[7]	34’h0	无效操作	/	/	/	

在 INST 得到了寻址的地址之后，我们将会按照如下的电路图依次从 CMU（就是从 cache）和 data_ram 中获取数据，如上文所述，我们一次访问 CMU 只需要 1 个周期，而一次访问 data_ram 则需要 18 个周期（data[5]的 18*2 是指我们首先需要一步访存将脏行写回，在一步访存将需要写的新数据换入。）

电路图：



另外，我们发现，简单起见，我们没有使用 data_r 的输出， CMU 的输出直接接地了，并不使用每次读取 CMU 的数值，同时，我们写入的值也比较随意（它和 clk_count，inst_count 有关，而不是和 CMU 内部的值有关）这样一来，我们检查的重点将是 clk_count ,时钟周期计数，hit_count，缓存命中计数（本次试验中应该是 3），有效指令计数（本次实验中应该是 7），假如需要观察读取的数据是否正确，那么需要打开 CMU 的内部单元观察 cache_out 和 data_r，我们的值和 PPT 比对也是正确的。

4．top 仿真模块

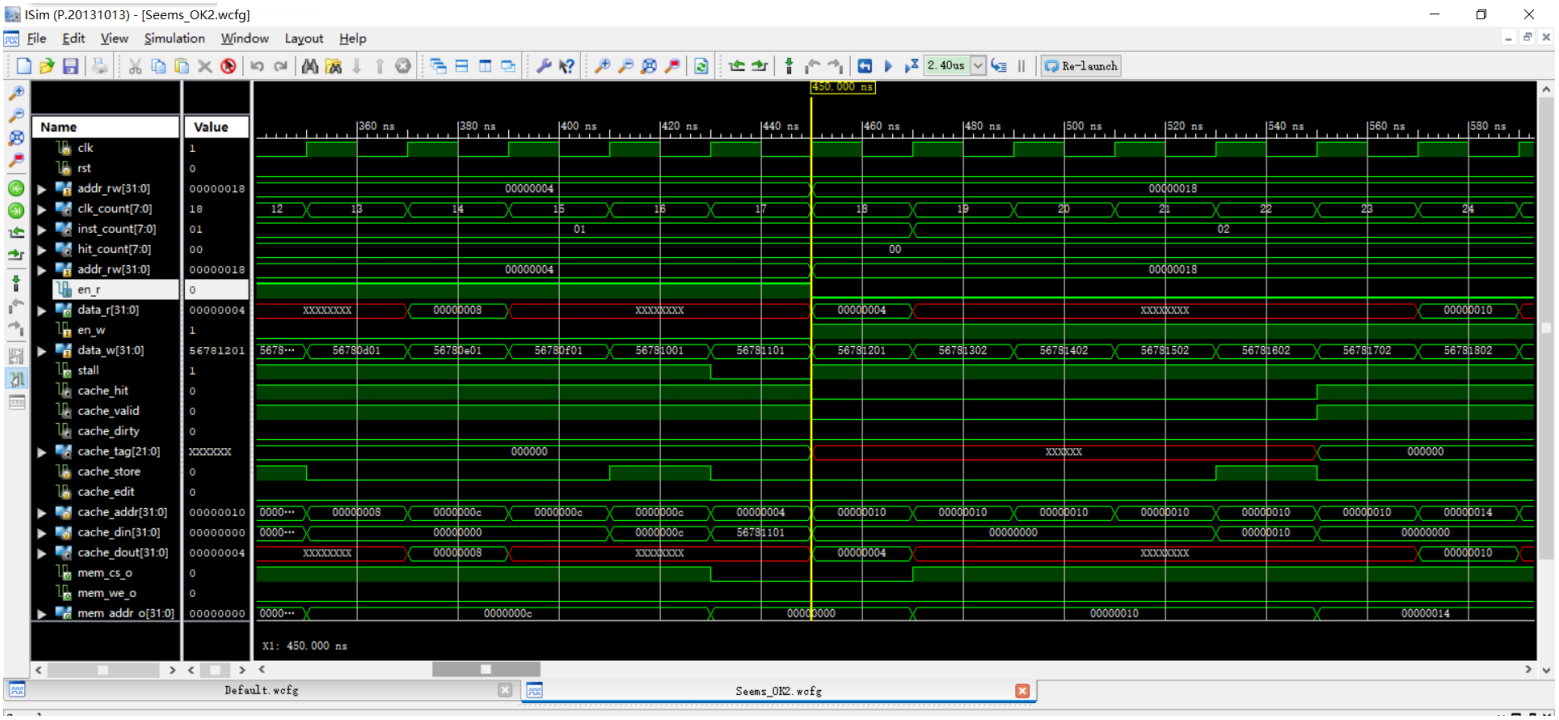
这里面的逻辑就比较简单了，因为 top 模块已经实现了按照时钟周期依次读取的 index 了，所以，top 只要给出一个初始化信号和若干时钟信号即可。

```
`timescale 1ns / 1ps

module sim_top2;

    // Inputs
    reg clk;
    reg rst;

    // Outputs
    wire [7:0] clk_count;
    wire [7:0] inst_count;
    wire [7:0] hit_count;

    // Instantiate the Unit Under Test (UUT)
    top2 uut (
        .clk(clk),
        .rst(rst),
        .clk_count(clk_count),
        .inst_count(inst_count),
        .hit_count(hit_count)
    );

    initial begin
        // Initialize Inputs
        clk = 0;
        rst = 1;

        // Wait 100 ns for global reset to finish
        // Wait 100 ns for global reset to finish
        #95 rst = 0;
```

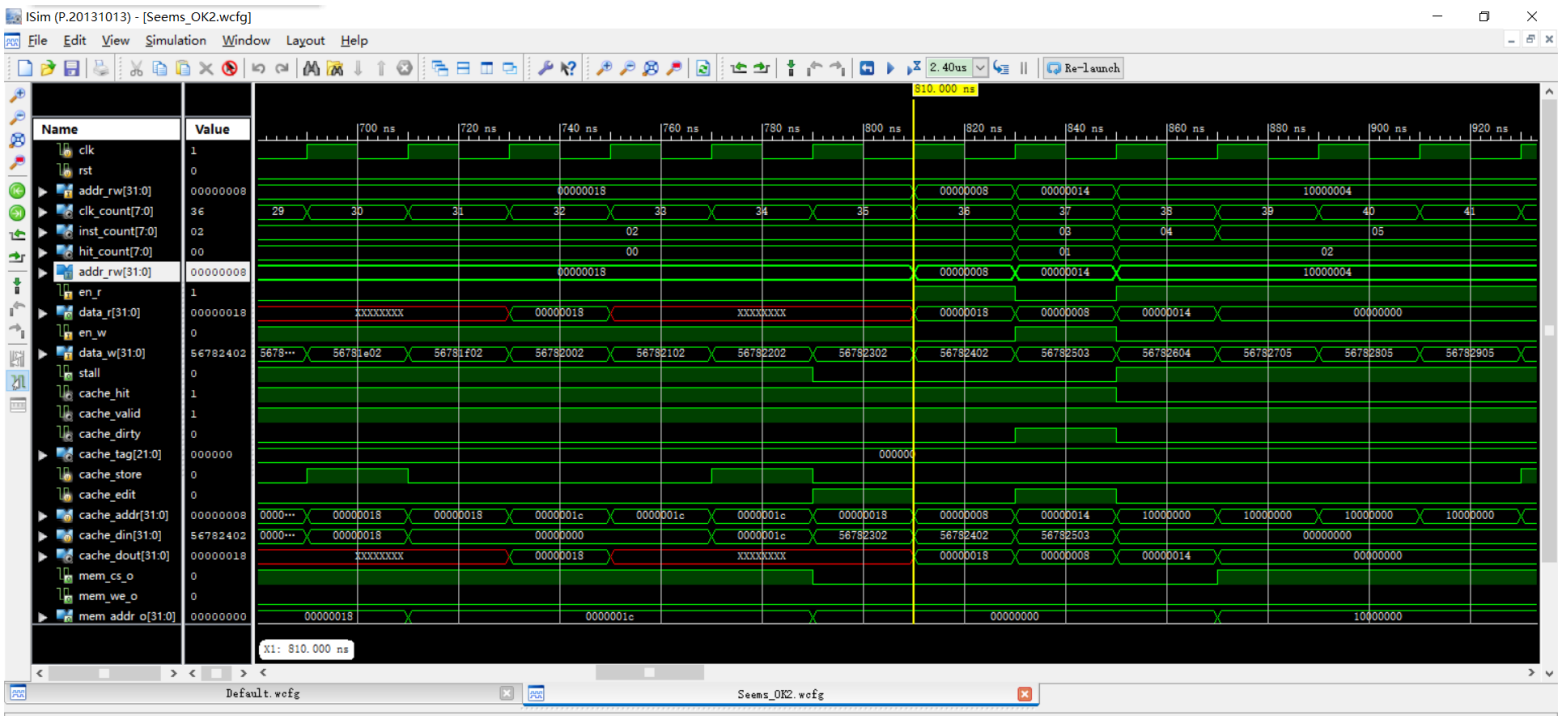
```
        // Add stimulus here

    end

    initial forever #10 clk = ~clk;

endmodule
```

5. 仿真图

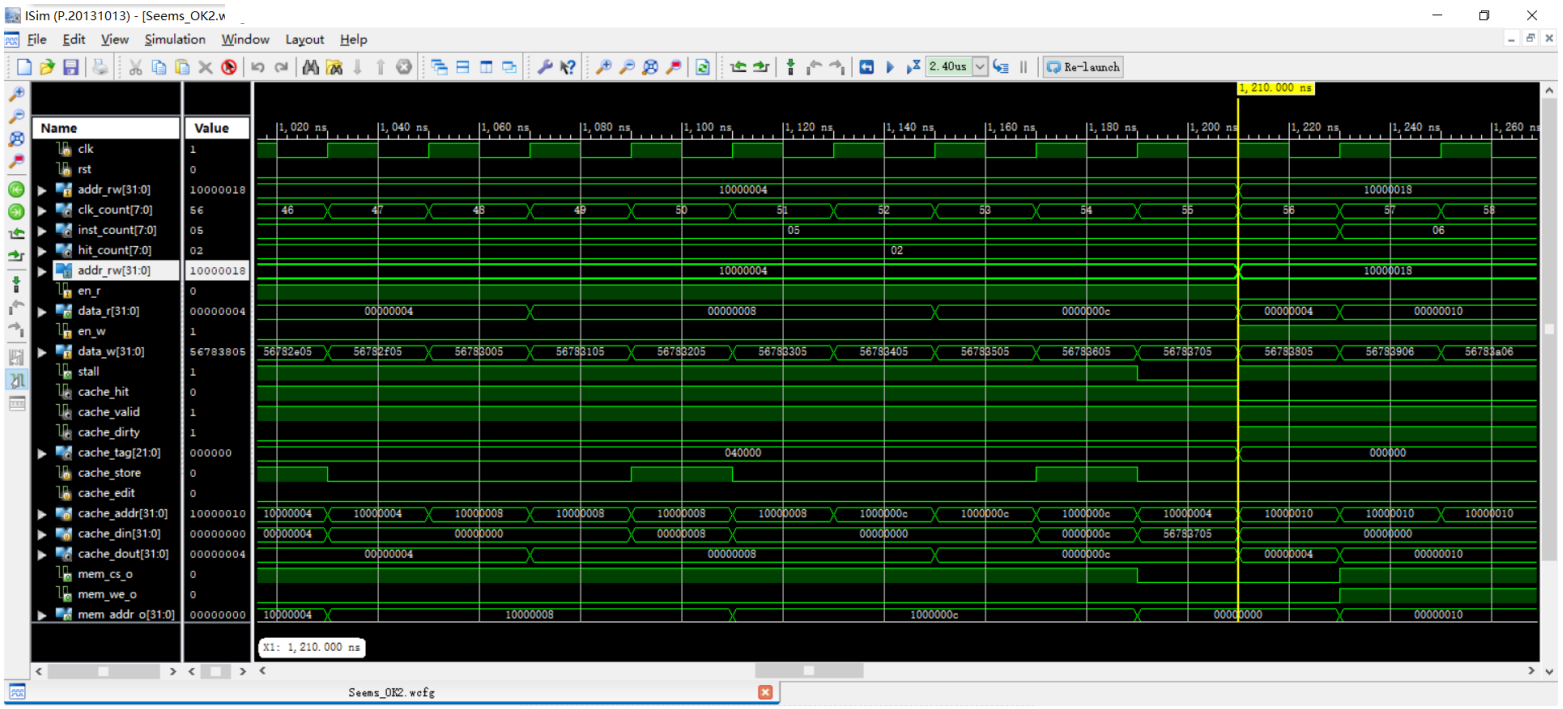
0000_0004 的读失配（耗时多个周期）：



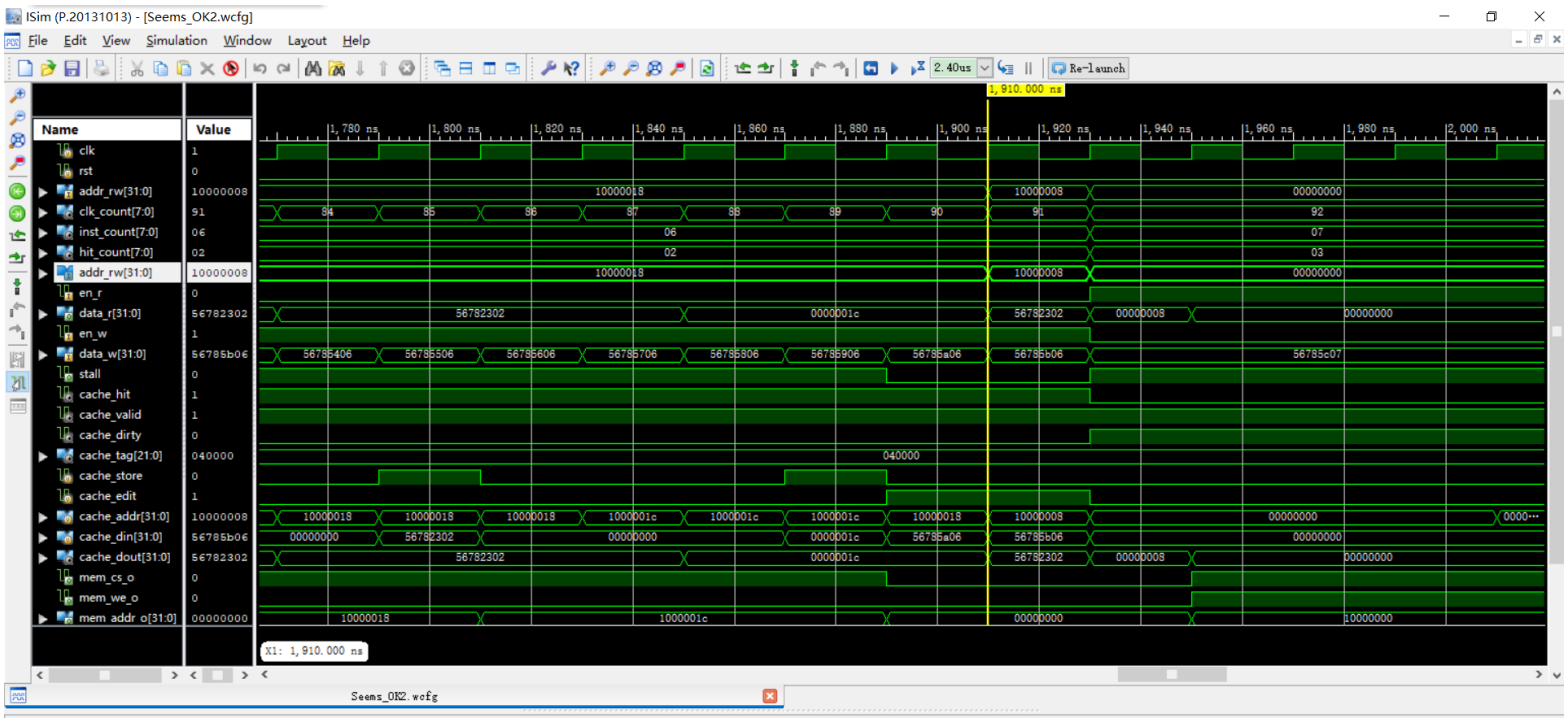
0000_0018 的写失配（耗时多个周期）；0000_0008 的读命中（一个周期）；0000_0014 的读命中（一个周期）



1000_0004 非脏行的读失配和换出



1000_0018 脏行的写失配和换出，以及 1000_0008 的写命中；



五、 讨论与心得

首先需要注意的是 2.5 中，dout 信号如果单纯按照

```
assign dout = ……;
```

或者

```
always @(*)begin
    dout = …… ;
end
```

这样的逻辑来处理的话，会产生这样的时序图：

1.在 din 有值的时候，dout 也会有值；

2. edit 信号产生之后，过了一个周期，而不是两个周期，就有 dout 的输出。

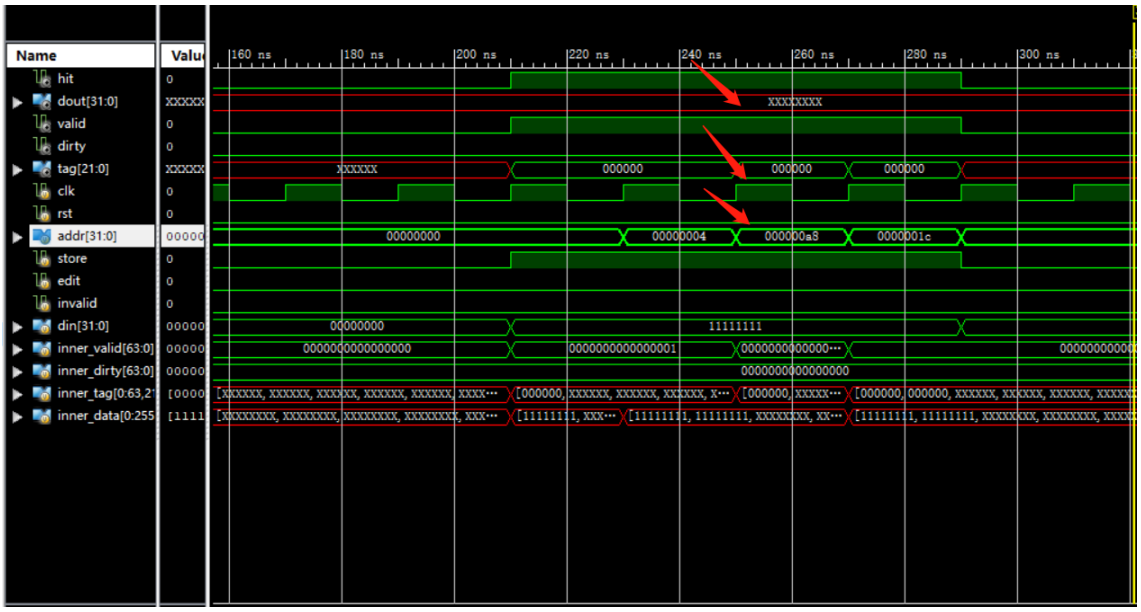
这些虽然看起来是小问题，因为第一个问题可以用

```
if(~load) begin
    dout <= ……;
end
```

这样的语句解决，防止 cacheout（缓存泄露），而第二个周期数目似乎也仅仅相差几个纳秒，但是因为不知道这些会不会影响后续实验（这个是 cache，我们怕他影响后续的 CMU），我们这里花费了很大的精力让他和 PPT 上的仿真图一模一样，我们的方法如下。

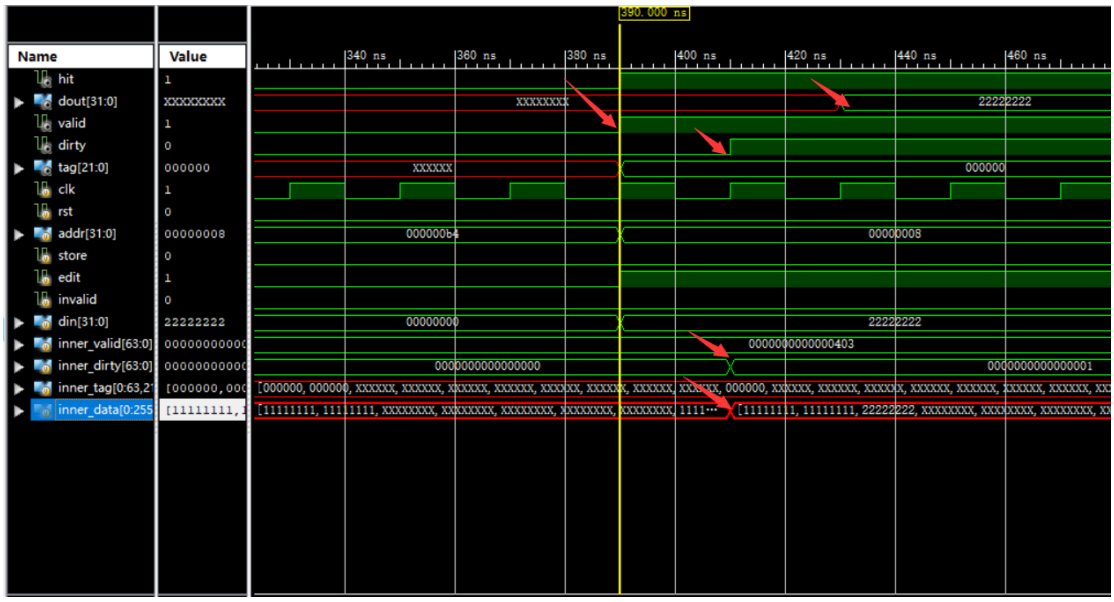
我们将 dout 放置在 `always @(posedge clk)begin..... end` 语句之中，和 inner_valid 那些信号一并处理，这样做的好处是，每个时钟上升沿，dout 都会根据上个周期的结果来判断是否应当有输出，所以如果有一个缓存行，在本周期被更新了，那么一定会等

到下一个时钟周期，他才会被 dout 读出来，否则就不会有读出的操作，（因为在 always @（posedge clk）begin end 语句块中，inner_data 的更新一直在 dout 之后，而 always 语句块室友时序的，所以我们可以实现这样的操作）。比如，下图中



我们在这个时钟上升沿 addr 产生了变化，而 dout 一直没有变化，因为 addr 先赋值给 inner_data，在这个上个时钟周期末完成，使得 inner_data[4]被赋值，这个时钟周期的上升沿，inner_data[a8]将会赋值给 dout，显然 inner_data[a8]尚未被写，于是 dout 先读出空值，然后再在这个时钟周期之后写 inner_data[a8]，下一个时钟周期倘若 addr 信号还是 a8，才可以读出来 dout。

同理，我们看到第二个判定点：



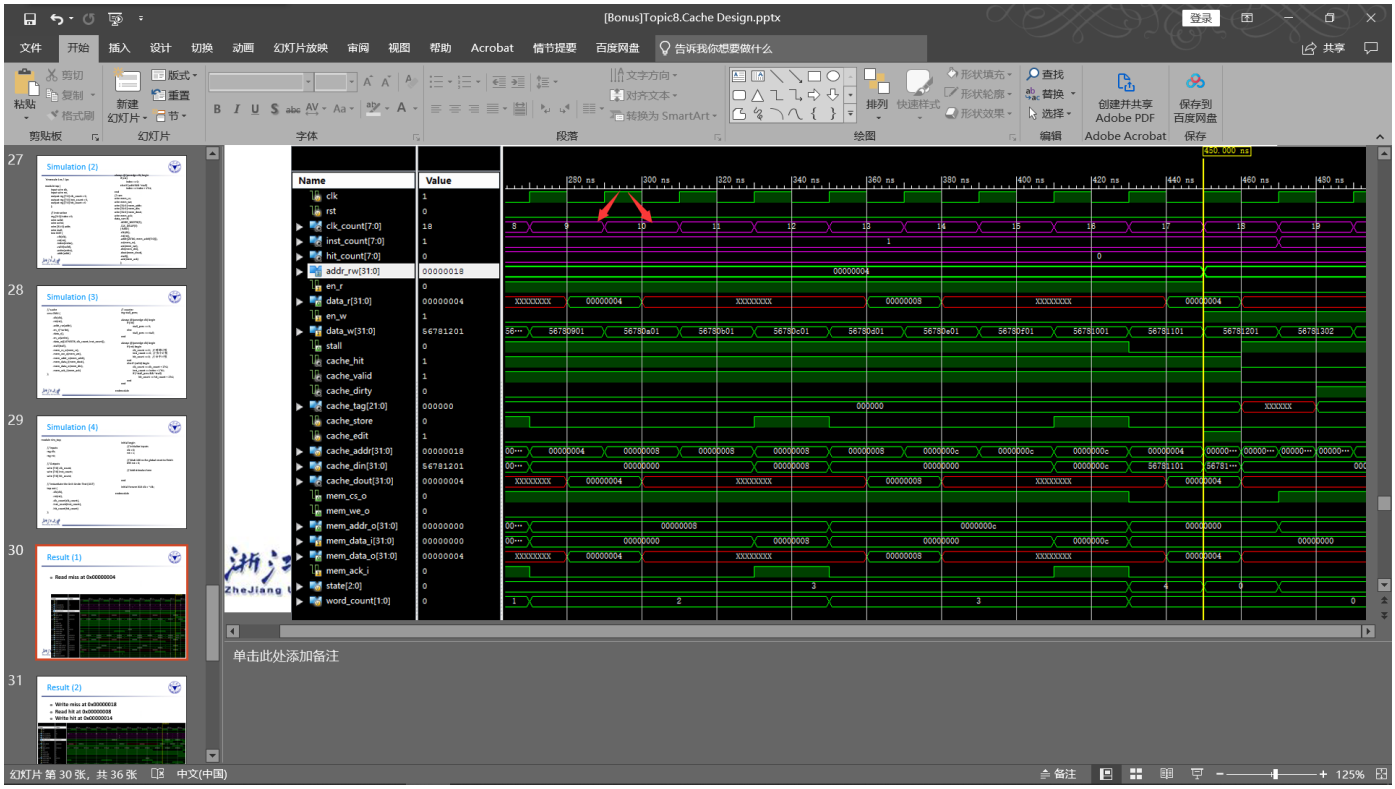
第一个时钟上沿因为读入的时候 hit 尚没有置位，所以我们没有修改 inner_dirty，也没有修改 dirty 这个输出信号（这是可以的，因为按照定义，我们至少要 edit 过一个

内存行之后他才可以是脏的），第二个时钟上沿，我们因为 hit 置位，所以我们可以修改 inner_valid，inner_tag 和 valid 输出信号，同时修改了 inner_data，于是在第三个时钟周期，我们就可以读出 dout。

在 CMU 的实验中，我们会有发现我们使用的是 cmu2 和 top2,因为我们第一个版本的 CMU 似乎有点问题，于是我们重构了一次。

另外，我们发现在第二个版本中，我们依旧有一个和 PPT 上不一样的地方，那就是我们的 cacheout 和 data_r 的读取比起 PPT 早半个周期，这可能是因为 PPT 是设置成了下降沿更新，而我们所有的信号变化都是默认上升沿更新。

最后，PPT 的图有一个陷阱，



他的进位制是 10 进制的，而不是惯例的 16 进制，所以一开始我们用 16 进位制的时候发现时钟周期有偏差，修改为 10 进制之后就完全相同了。